A Data-centric Profiler for Parallel Programs

Xu Liu and John Mellor-Crummey
Department of Computer Science
Rice University
Houston, TX, USA
{xl10, johnmc}@rice.edu

ABSTRACT

It is difficult to manually identify opportunities for enhancing data locality. To address this problem, we extended the HPCToolkit performance tools to support data-centric profiling of scalable parallel programs. Our tool uses hardware counters to directly measure memory access latency and attributes latency metrics to both variables and instructions. Different hardware counters provide insight into different aspects of data locality (or lack thereof). Unlike prior tools for data-centric analysis, our tool employs scalable measurement, analysis, and presentation methods that enable it to analyze the memory access behavior of scalable parallel programs with low runtime and space overhead. We demonstrate the utility of HPCToolkit's new data-centric analysis capabilities with case studies of five well-known benchmarks. In each benchmark, we identify performance bottlenecks caused by poor data locality and demonstrate non-trivial performance optimizations enabled by this guidance.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Measurement techniques, Performance attributes.

General Terms

Performance, Measurement.

Keywords

Data-centric profiling, scalable profiler, data locality.

1. INTRODUCTION

In modern processors, accesses to deep layers of the memory hierarchy incur high latency. Memory accesses with high latency not only degrade performance but also increase energy consumption. For many programs, one can reduce average memory latency by staging data into caches and accessing it thoroughly before it is evicted. Access patterns that do so are said to have excellent data locality. There are two types of data locality common to both sequential and multithreaded programs: *spatial* and *temporal*. An access pattern exploits *spatial* locality when it accesses a memory location and then accesses nearby locations soon afterward. Typically, spatial locality goes unexploited when accessing data with a large stride or indirection. An access pattern exhibits *temporal* locality when it accesses individual memory locations multiple times.

Multi-socket systems have an extra layer in the memory hierarchy that poses an additional obstacle to performance. Processors in such systems are connected with a high bandwidth link, e.g., HyperTransport for AMD processors and QuickPath for Intel processors. To provide high aggregate memory bandwidth on multi-socket systems, memory is partitioned and some is directly attached to each processor. Each processor has its own memory controller to access directly-attached memory. Such an architecture has Non-Uniform Memory Access (NUMA) latency. Accesses to directly-attached memory are known as *local* accesses. A processor can also access memory attached to other processors; such accesses are known as *remote* accesses. Remote accesses have higher latency than local accesses, so each thread in a multi-threaded program must access local data for high performance. A thread that primarily performs local accesses is said to have *NUMA* locality.

Tools for identifying data locality bottlenecks use either *simulation* or *measurement*. Simulation-based data-centric tools, such as CPROF [23], MemSpy [27], ThreadSpotter [33], SLO [6, 5], and MACPO [32], instrument some or all memory accesses and compute the approximate memory hierarchy response with a cache simulator. There are two drawbacks to simulation. First, gathering information about memory accesses with pervasive instrumentation is expensive. Although sampling techniques for monitoring accesses, e.g., [34, 39], can reduce measurement overhead for instrumentation, they do so with some loss of accuracy. Second, the accuracy of simulation-based methods depends on the cache simulators they use. Since it is prohibitively expensive for cache simulators to model all details of memory hierarchies in modern multi-socket systems, the aforementioned tools make simplifying assumptions for simulators that reduce the tools' accuracy.

Unlike simulation-based tools, measurement-based tools

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

SC'13, November 17–22, 2013, Denver, CO, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2378-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2503210.2503297

collect performance metrics using hardware performance counters. Measurement-based tools can directly assess the performance of a program execution with low overhead. Measurement-based tools can be classified as code-centric and/or data-centric. In the best case, *code-centric* tools such as VTune [13], Oprofile [24], CodeAnalyst [3], and gprof [12] map performance metrics back to statements accessing data. Such code-centric information is useful for identifying problematic code sections but fails to highlight problematic variables. *Data-centric* tools, on the other hand, attribute performance metrics to variables or dynamic memory allocations. A tool that combines both data-centric and code-centric analysis is more powerful for pinpointing and diagnosing data locality problems. Section 2 elaborates on the motivation for data-centric tools.

A variety of data-centric tools currently exist; we discuss them in detail in Section 6. Some tools focus on data locality in sequential codes [25, 26, 7]; the others focus on NUMA problems in threaded codes [28, 20]; none of them supports comprehensive analysis of all kinds of data locality problems. Moreover, existing tools work on modest numbers of cores on a single node system; none of them tackles the challenge of scaling and is applicable across a cluster with many hardware threads on each node. Obviously, data-centric measurement tools must be scalable if they are to be used to study codes on modern supercomputers with non-trivial input data.

To address this challenge, we extended the HPCToolkit performance tools [2] with data-centric capabilities to measure and analyze program executions on scalable parallel systems. Our resulting tools have three unique capabilities for data-centric measurement and analysis.

- They report all data locality issues. HPCToolkit can measure and analyze memory latency in threaded programs running on multi-socket systems. Failing to exploit temporal, spatial, and NUMA data locality exacerbates memory latency.
- They work for large-scale hybrid programs that employ both MPI [29] and OpenMP [31]. HPCToolkit collects data-centric measurements for scalable parallel programs with low runtime and space overhead. Moreover, HPCToolkit aggregates measurement data across threads and processes in a scalable way.
- They provide intuitive analysis of results for optimization. HPCToolkit provides an intuitive graphical user interface that enables users to analyze data-centric metrics attributed to variables, memory accesses, and full calling contexts. It provides multiple data-centric views to highlight variables and attribute costs to instructions that access them.

HPCToolkit exploits hardware support for data-centric measurement on both AMD Opteron and IBM POWER processors. To evaluate the effectiveness of our methods, we employ our tools to study five parallel benchmarks, covering MPI, OpenMP and hybrid programming models. With the help of HPCToolkit, we identified data locality problems in each of these benchmarks and improved their performance using insights gleaned with our tools.

The rest of this paper is organized as follows. Section 2 elaborates on the motivation for this work. Section 3 describes the hardware support that HPCToolkit uses for data- and code-centric analysis. Section 4 presents the design and implementation of our new capabilities in HPCToolkit. Section 5 studies five well-known benchmarks and shows how HPCToolkit's data-centric analysis capabilities provide insights about data access patterns that are useful for tuning. Section 6 describes previous work on data-centric tools for analyzing data locality problems and distinguishes our approach. Finally, Section 7 summarizes our conclusions and outlines our future directions.

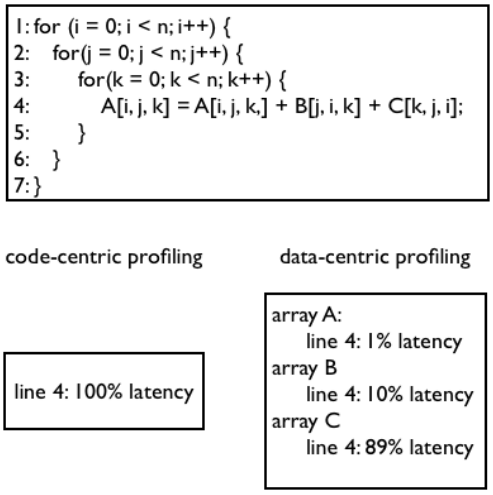
1: for (i = 0; i < n; i++) {
2: for(j = 0; j < n; j++) {
3: for(k = 0; k < n; k++) {
4: A[i, j, k] = A[i, j, k,] + B[j, i, k] + C[k, j, i];
5: }
6: }
7: }
code-centric profiling
data-centric profiling
line 4: 100% latency
array A:
line 4: 1% latency
array B
line 4: 10% latency
array C
line 4: 89% latency

Figure 1: Code-centric profiling aggregates metrics for memory accesses in the same source line; data-centric profiling decomposes metrics by variable.

2. MOTIVATION

There are two principal motivations for extending HPCToolkit to support data-centric measurement and analysis of program executions on scalable parallel systems. In Section 2.1, we illustrate the importance of data-centric profiling for analyzing data locality problems. In Section 2.2, we describe why scalability has become an important concern for data-centric profilers.

2.1 Data-centric profiling

Two capabilities distinguish data-centric profiling from code-centric profiling. First, while code-centric profiling can pinpoint a source code line that suffers from high access latency, without a mapping from machine instructions to character positions on a source line (which to our knowledge hasn't been available since DEC's Alpha compilers), code-centric profilers can't distinguish the contribution to latency associated with different variables accessed by the line. In contrast, data-centric profiling can decompose the latency and attribute it to individual variables accessed by the same source line. Figure 1 illustrates the latency decomposition that data-centric profiling can provide. Data-centric methods can decompose the latency associated with line 4 and attribute it to different variables. From the percentage of latency associated, one can see that array `C` is of principal interest for data locality optimization. One can then continue to investigate the program to determine how to improve data locality for `C`. Second, data-centric profiling aggregates metrics from all memory accesses that are

```
for (i = 0; i < 100; i++) {
    var[i] = malloc(size);
}
```

Figure 2: A loop allocating data objects in the heap.

associated with the same variable. Aggregate metrics can highlight problematic variables in a program. This can help one identify when a data layout is poorly matched to access patterns that manipulate the data, or pinpoint inefficient memory allocations on a NUMA architecture.

2.2 Scalability

Scalability is a significant concern for data-centric profilers because (a) application performance on scalable parallel systems is of significant interest to the HPC community; (b) memory latency is a significant factor that limits performance and increases energy consumption; and (c) it is desirable to study executions on large-scale data sets of interest rather than forcing application developers to construct representative test cases that can be studied on small systems. Unfortunately, existing data-centric profilers do not scale to highly-parallel MPI+OpenMP programs.

Data-centric profilers for scalable parallel systems should have low time and space overhead. Runtime overhead mainly comes from two parts: collecting performance metrics and tracking variable allocations. One can reduce the overhead of collecting metrics by sampling hardware performance monitoring units (described in Section 3) with a reasonable sampling period. However, tracking variable allocations is difficult to control. If a program frequently allocates data, the overhead of tracking allocations can be unaffordable. Unfortunately, existing tools lack the capability to reduce such overhead; instead, they assume that programs don't frequently allocate heap data. We show a benchmark in our case study that contradicts this assumption.

Space overhead is also critical when profiling parallel executions. Systems like LLNL's Sequoia support millions of threads. If each thread in a large-scale execution generates 1MB of performance data, a million threads would produce a terabyte of performance data. Thus, for analysis at scale a compact profile is necessary to keep the size of measurement data manageable. However, many existing tools trace variable allocations and memory accesses [16, 20]. The size of memory access traces is proportional to execution time and the number of active threads. A trace of variable allocations can easily grow unaffordably large. Consider the code in Figure 2. A memory allocation is called 100 times in a loop, so 100 allocations would be recorded in a thread trace. However, if this loop is in an OpenMP parallel region executed by each MPI process, millions of allocations would be recorded in a large-scale execution on a system like LLNL's Sequoia.

Besides overhead, existing tools do not display data-centric results in a scalable way for effective analysis. Again, consider the code in Figure 2. Metrics may be dispersed among 100 allocations, without showing any hot spot. However, one might be interested in aggregating metrics for these 100 allocations per thread. If this loop is called by multiple threads and MPI processes, metrics associated with these data objects should be coalesced to highlight `var` as a problematic array.

To address these issues, we added scalable data-centric profiling support to the HPCToolkit performance tools. HPCToolkit employs novel techniques to scale the data collection and presentation with low time and space overhead.

3. HARDWARE SUPPORT FOR DATA-CENTRIC PROFILING

To support accurate measurement and attribution of performance metrics in out-of-order processors, Dean et al. [9] developed instruction-based sampling (IBS). When using IBS, a performance monitoring unit (PMU) periodically selects an instruction for monitoring. As a monitored instruction moves through the execution pipeline, the PMU records information about the occurrence of key events (e.g., cache and TLB misses), latencies, and the effective address of a memory operand. When a sampled instruction completes, the PMU triggers an interrupt, signaling that details about a monitored instruction's execution are available, along with the address of the monitored instruction. The first tool using IBS was DEC's Dynamic Continuous Profiling Infrastructure (DCPI)—a low overhead, system-wide, flat profiler [4].

Recent AMD Opteron processors (family 10h and successors) support instruction-based sampling [10]. Using IBS, one can analyze the execution of instructions that complete. The PMU records detailed information about an instruction as it executes, including the effective address (virtual and physical) for a load/store, details about the memory hierarchy response (e.g., cache and TLB miss or misalignment), and various latency measures.

IBM POWER5 and successors use a mechanism similar to IBS to count marked events [35]. When a marked event is counted, POWER processors set two special purpose registers. The sampled instruction address register (SIAR) records the precise instruction address of the sampled instruction. The sampled data address register (SDAR) records the effective address touched by a sampled instruction if this instruction is memory related. A PMU can be configured to trigger an interrupt when a marked event count reaches some threshold. When a sample is triggered, HPCToolkit can use SIAR and SDAR to attribute marked events to both code and data.

Other processors have data-centric measurement capabilities as well. Intel Nehalem processors and their successors support precise event-based sampling [14]. Event Address Registers (EAR) on Itanium processors also support data-centric measurement [15].

Common features for data-centric measurement are (1) support for sampling memory accesses, (2) one or more mechanisms for assessing the cost of an access in terms of performance events or access latency, and (3) support to identify a precise instruction pointer (IP) and an effective data address for a sampled access. We use events or latency to quantify access costs, the precise IP to associate costs with memory accesses, and the effective address to associate costs with variables.

4. DATA-CENTRIC CAPABILITIES FOR HPCTOOLKIT

Figure 3 shows a simplified workflow that illustrates data-centric measurement and analysis in HPCToolkit. HPCToolkit consists of three principal components: an online

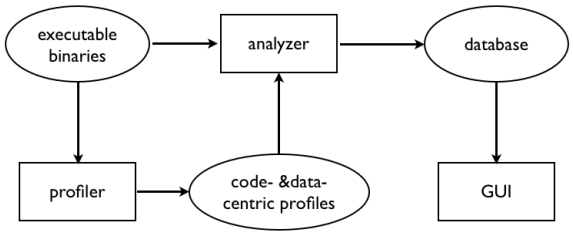

Figure 3: A simplified workflow that illustrates data-centric measurement and analysis in HPCToolkit. Rectangles are components of HPCToolkit; ellipses are inputs or outputs of different components.

call path profiler, a post-mortem analyzer, and a graphical user interface (GUI) for presentation. The profiler takes a fully optimized binary executable as its input. For each thread in a program execution, the profiler collects samples and associates costs with memory accesses and data objects. Section 4.1 describes the profiler. The post-mortem analyzer, described in Section 4.2, gathers all profiles collected by the profiler for each process and thread. It also analyzes the binaries, extracts information about static code structure, and maps profiles to source lines, loops, procedures, dynamic calling contexts, and variable names. Finally, the GUI, also described in Section 4.2, displays intuitive views of data-centric analysis results that highlight problematic data and accesses. Each component is designed to scale.

4.1 Online call path profiler

HPCToolkit's call path profiler is loaded into a monitored program's address space at link time for statically linked executables or at runtime using LD_PRELOAD for dynamically linked executables. As a program executes, HPCToolkit's profiler triggers samples, captures full calling contexts for sample events, tracks variables, and attributes samples to both code and variables. To minimize synchronization overhead during execution, each thread records its own profile. The following four sections describe the implementation of data-centric support for each of these capabilities.

4.1.1 Triggering samples

The profiler first programs each core's PMU to enable instruction-based sampling or marked event sampling with a pre-defined period. When a PMU triggers a sample event, the profiler receives a signal and reads PMU registers to extract performance metrics related to the sampled instruction. To map these performance metrics to both code and data, the profiler records the precise instruction pointer of the sampled instruction and the effective address of the sampled instruction if it accesses memory.

4.1.2 Capturing full calling contexts

HPCToolkit unwinds the call stack at each sample event. To do so, it uses on-the-fly binary analysis to locate the return address for each procedure frame on the call stack [38]. Call paths are entered into a calling context tree (CCT). The root of a CCT represents *main* or a thread start function; internal nodes represent call sites; and leaves represent instructions where samples were triggered. A CCT reduces the space needed for performance data by coalescing common call path prefixes.

Supporting data-centric analysis required two changes to HPCToolkit's call stack unwinder. First, we adjust the leaf node of the calling context we obtain by unwinding from the signal context to use the precise IP recorded by PMU hardware. This avoids "skid" between the monitored instruction and the sample event that typically occurs on out-of-order processors. Second, we create multiple CCTs for each thread. Inserting a sample call path into a specific CCT depends on the sample feature. For example, we create a CCT that includes all samples that do not access memory; other CCTs include call paths that touch different types of variables, such as static and heap-allocated variables.

4.1.3 Tracking variables

To support data-centric analysis, we augmented HPCToolkit to track the life cycle of each variable. We track the allocation and deallocation of static and heap allocated data. Variables that do not belong to static or heap allocated data are treated as unknown data.

Static data.

Data allocated in the *.bss* section in load modules are static data. Each static variable has a named entry in the symbol table that identifies the memory range for the variable with an offset from the beginning of the load module. The life cycle of static variables begins when the enclosing load module (executable or dynamic library) is loaded into memory and ends when the load module is unloaded. The profiler tracks the loading and unloading of load modules. When a load module is loaded into memory, HPCToolkit reads the load module's symbol table to extract information about the memory ranges for all of its static variables. These memory ranges are inserted into a map for future use. All load modules in use are linked in a list for easy traversal. If a load module is unloaded, the load module together with its search tree of static data is removed from the list.

HPCToolkit tracks all static variables used during a program execution. Unlike other tools, HPCToolkit not only tracks static variables in the executable, but also static variables in dynamically-loaded shared libraries. Moreover, it collects fine-grained information for each static variable rather than simply attributing metrics to load modules.

Heap-allocated data.

Variables in the heap are allocated dynamically during execution by one of the *malloc* family of functions (malloc, calloc, realloc). Since heap-allocated data may be known by different aliases, e.g., function parameters, at different points in an execution, HPCToolkit's profiler uses the full call path of the allocation point for a heap-allocated data block to uniquely identify it throughout its lifetime. To associate a heap-allocated variable with its allocation call path, the profiler wraps memory allocation and free operations in a monitored execution. At each monitored allocation, the profiler enters into a map an association between the address range of a data block and its allocation point. At each monitored free, the profiler deletes an association from the map.

Unknown data.

There are variables that do not belong to static or heap allocated data. Such variables are either not easily tracked or have little impact for performance. For example, C++ tem-

plate containers directly use a low level system call `brk` to allocate memory. Instead of returning the allocated ranges, it sets the data segment. For that reason, HPCToolkit does not track C++ template container allocations. HPCToolkit also treats stack variables as unknown data because stack variables seldom become data locality bottlenecks.

Recording address range information for static variables incurs little overhead because it happens once when a load module is loaded into the program's address space. However, the overhead of tracking heap allocations and deallocations hurts the scalability of the profiler. If a program allocates and frees memory with high frequency, wrapping allocates and frees and capturing the full calling context for each allocation may cause large overhead. For example, the execution time of AMG2006, one of our case study benchmarks, increases by 150% when monitoring all allocations and frees. To reduce such overhead, we use the following three strategies.

- We do not track all memory allocations. Usually, large arrays suffer from more severe locality problems than small ones. Typically, there are also more opportunities for optimizing data locality for large arrays. For that reason, HPCToolkit does not track any heap allocated variable smaller than a threshold, which we have chosen as 4K. However, we still track all calls to free to avoid attributing costs to wrong variables. Because we don't collect calling contexts for frees, wrapping all frees is not costly.
- We use inlined assembly code to directly read execution context information from registers to aid in call stack unwinding. Our assembly code incurs lower overhead than *libc*'s `getcontext`.
- Since unwinding the call stack for frequent allocations in deep calling contexts is costly, we reduce unwinding overhead by identifying the common prefix for adjacent allocations to avoid duplicate unwinds of prefixes that are already known. We accomplish this by placing a marker (known as a trampoline) in the call stack to identify the least common ancestor frame of the calling contexts for two temporally adjacent allocations [11]. Using this technique, each allocation only needs to unwind the call path suffix up to the marked frame.

Because the first method can lead to inaccuracy due to the information loss, we only use it when necessary to avoid unaffordable time overhead. We always enable the other two methods because they are always beneficial. In our case study of AMG2006, these approaches reduce the time overhead of tracking variables from 150% to less than 10%.

4.1.4 Attributing metrics to variables

By correlating information about accesses from PMU samples with memory ranges for variables, HPCToolkit performs data-centric attribution on-the-fly. It first creates three CCTs in each profile, each recording a different storage class: static, heap, and unknown. This aggregation highlights which storage class has more problematic locality.

For each sample, HPCToolkit searches the map of heap allocated variables using the effective address provided by the PMU registers. If a sample in a thread touches a heap allocated variable, the thread prepends the call path for the variable allocation to the call path for the sample and then adds it to its local heap data CCT. It is worth noting that the memory allocation call path may reside in a different thread than the one the PMU sample takes for an access. Copying a call path from another thread doesn't require a lock because once created, a call path is immutable. If a copied call path is already in the local thread CCT, the thread coalesces the common paths. Although a memory access is mapped to a heap allocated variable using the allocated memory ranges, the allocation call path uniquely identifies a heap allocated variable. This CCT copy-and-merge operation successfully addresses the problem of multiple allocations with the same call path, which we illustrated with Figure 2. If multiple heap allocated data objects have the same allocation call path, they are merged online and treated as a single variable. Memory accesses to different variables are separated into different groups identified by the variables' allocation paths.

If HPCToolkit does not find a heap allocated variable matching an access, it looks up the effective address for the access in data structures containing address ranges for static variables. The search is performed on each load module (the executable and dynamically loaded libraries) in the active load module list to look for a variable range. If the sample accesses a static variable, the profiler records the variable name from the symbol table in a dummy node and inserts it into the static data CCT of the thread that takes the sample. The sample with the full call path is inserted under the dummy node. Therefore, if multiple samples touch the same static variable, they all have the same dummy node as their common prefix for merging. Like heap allocated variables, the dummy nodes of static variable names separate the CCT into multiple groups.

If the sample does not access any heap allocated variable or static variable, we insert the sample to the thread's unknown data CCT. All of a thread's call paths touching unknown data are grouped together in this CCT.

The data-centric attribution strategy used by HPCToolkit's profiler aggregates data access samples in both coarse-grain (storage class) and fine-grain (individual variables) ways. Because each thread records data into its own CCTs, no thread synchronization is needed and data-centric attribution incurs little overhead.

4.2 Post-mortem analyzer and user interface

HPCToolkit's post-mortem analyzer takes load modules and profiles as inputs. It reads the symbol table from the executable and dynamically loaded libraries, and maps CCT nodes to either function or variable symbols. It then extracts line mapping information from debugging sections and maps symbol names to source code.

To generate compact profile results, which is important for scalability, HPCToolkit's post-mortem analyzer merges profiles from different threads and processes. Data-centric profiles, which consist of at most three CCTs per thread (one each for static, heap, and unknown storage classes), are amenable to coalescing. The analyzer merges CCT's of the same storage class across threads and processes. Context paths for individual variables and their memory accesses can be merged recursively across threads and processes. For heap allocated variables, if their allocation call paths are the same, even they are from different threads or processes, they are coalesced. For static variables, the analyzer merges them from different threads and processes if they have the same symbol name. Because all memory access call paths

Table 1: Measurement configuration and overhead of benchmarks

code	number of cores	monitored events	execution time	execution time with profiling
AMG2006	4 MPI processes, 128 threads/process	PM_MRK_DATA_FROM_RMEM[1]	551s	604s (+9.6%)
Sweep3D	48 MPI processes, no threads	AMD IBS	88s	90s (+2.3%)
LULESH	48 threads	AMD IBS	17s	19s (+12%)
Streamcluster	128 threads	PM_MRK_DATA_FROM_RMEM	25s	27s (+8.0%)
NW	128 threads	PM_MRK_DATA_FROM_RMEM	77s	80s (+3.9%)

have variable CCT nodes as prefixes, they can be automatically merged after the merging of variables. The profile merging overhead grows linearly with the increasing number of threads and processors used by the monitored program. HPCToolkit's post-mortem analyzer uses a scalable MPI-based algorithm to parallelize the merging process using a reduction tree [36].

Finally, the analyzer outputs a database for HPCToolkit's GUI. The GUI sorts performance metrics related to each variable and instruction. It provides a top-down view to explore the costs associated with each dynamic calling context in an execution. One can easily identify performance losses within full calling contexts for variables or instructions. Moreover, HPCToolkit's GUI also provides a complementary bottom-up view. If the same `malloc` function is called in different contexts, the bottom-up view aggregates all performance losses and associates them with the malloc call site. In case studies, we show how these two views guide data locality optimization.

5. CASE STUDIES

We evaluated HPCToolkit's data-centric extensions for analyzing data locality issues on two machines. Our study focused on evaluating measurement and analysis of highly multithreaded executions at the node level. HPCToolkit's MPI-based post-mortem analysis naturally scales for analysis of data from many nodes.

The first test platform for our study is a POWER7 cluster. Each node has four POWER7 processors with a total of 128 hardware threads. Each POWER7 processor in a node has its own memory controller so there are four NUMA nodes. We reserved 4 nodes, up to 512 threads to evaluate the scalability of HPCToolkit. The applications we studied use one or both of MPI for inter-node communication and OpenMP for intra-node communication. A second test platform for evaluating HPCToolkit's data-centric analysis is a single node server with four AMD Magny-Cours processors. There are 48 cores in this machine and 8 NUMA locality domains.

We studied five well-known application benchmarks coded in C, C++ and Fortran, covering OpenMP, MPI and hybrid programming models. We built these programs using GNU or IBM compilers with full optimization. The programs we studied are the following:

- AMG2006 [22], one of LLNL Sequoia benchmarks, is a parallel algebraic multi-grid solver for linear systems arising from problems on unstructured grids. The driver for this benchmark builds linear systems for various 3D problems. It consists of three phases: initialization, setup and solver. AMG2006 is a MPI+OpenMP benchmark written in C.
- Sweep3D [1] is an ASCI benchmark that solves a time independent discrete ordinates 3D Cartesian geometry neutron transport problem. It is written in Fortran and parallelized with MPI without threading.
- LULESH [21], a UHPC application benchmark developed by Lawrence Livermore National Laboratory, is an Arbitrary Lagrangian Eulerian code that solves the Sedov blast wave problem for one material in 3D. In this paper, we study a highly-tuned LULESH implementation written in C++ with OpenMP.
- Streamcluster [8] is one of Rodinia benchmarks. For a stream of input points, it finds a predetermined number of medians so that each point is assigned to its nearest center. The quality of the clustering is measured by the sum of squared distances metric. It is written in C++ with OpenMP.
- Needleman-Wunsch (NW) [8], another Rodinia benchmark, is a nonlinear global optimization method for DNA sequence alignments. It is also implemented in C++ with OpenMP.

These benchmarks are configured to scale to the full size of our testbeds. The configuration and measurement overhead for these benchmarks is shown in Table 1. One may ask how we chose events to monitor and why these events affect performance. HPCToolkit either computes derived metrics [26] to identify whether a program is memory-bound enough for data locality optimization or counts occurrences of a specific event with a traditional hardware counter and evaluates its performance impact. We only apply data-centric analysis to memory-bound programs. As the table shows, in our case studies HPCToolkit's measurement overhead was 2.3–12%. The profile size (i.e., space overhead) ranged from 8–33 MB.

5.1 AMG2006

Figure 4 is an annotated top-down view from HPCToolkit's GUI for AMG2006, which executes with four MPI processes and 128 threads per process on our POWER7 cluster. The GUI consists of three panes. The top one shows the source code of a monitored program; the navigation pane at the bottom left shows program contexts such as routines, loops and statements. To present data-centric measurements, we add variable names or allocation call stacks as context prefixes in the navigation pane. The bottom right pane shows the metrics related to the context at the left. The GUI computes inclusive and exclusive values for all metrics. In the figures for our case studies, we show only inclusive metrics.

In Figure 4, the second line of the metric pane shows that 94.9% of the remote memory accesses are associated with heap allocated variables. Data allocated on line 175 of `hyper_CAlloc$AF6_3` is the target of 22.2% of the remote

[1]This is a marked event that measures data access from remote memory.

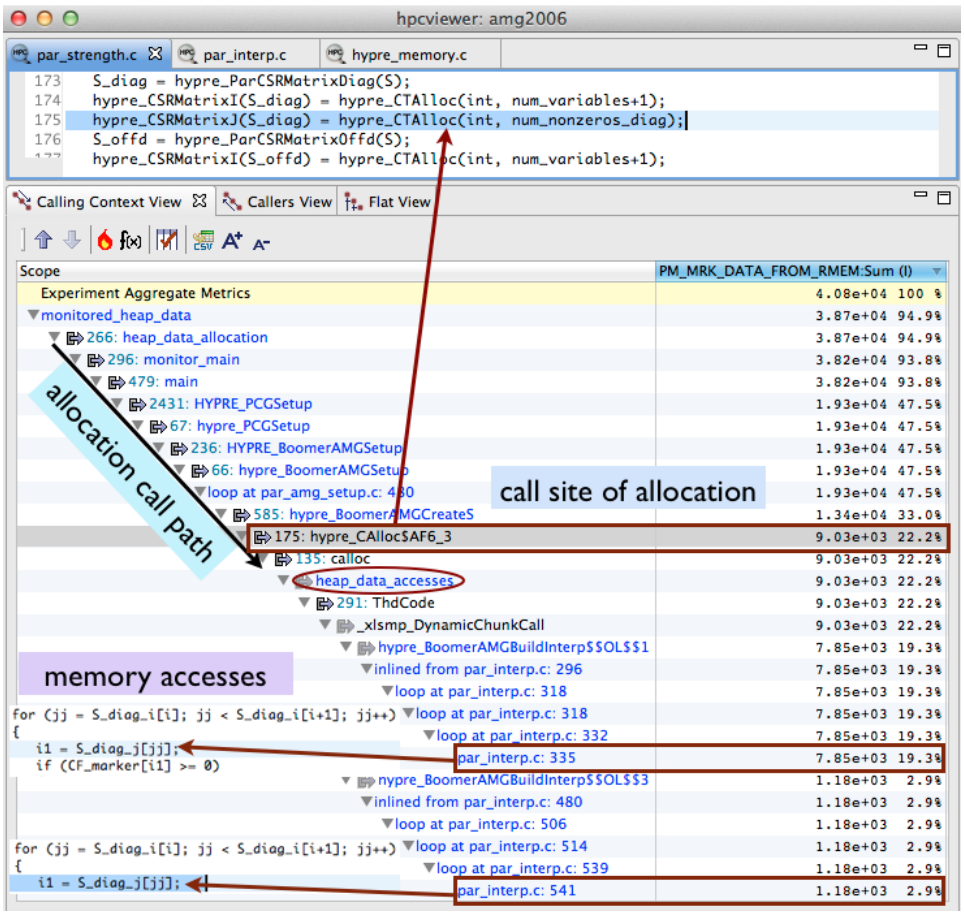

Figure 4: The top-down data-centric view of AMG2006 shows a problematic heap allocated variable and its two accesses that suffer most remote events to this variable.

memory accesses in the execution. Selecting the allocation call site in the navigation pane shows the source code for the allocation, enabling one to identify the variable allocated. The highlighted source code line shows that the column indices for non-zeros in a matrix (`S_diag_j`) are the target of these remote accesses. Deeper on the allocation call path, one can see that this variable is allocated by calloc. Immediately below the calloc is a dummy node *heap_data_accesses*; this node serves as the root of all memory accesses to this data. Calling contexts for two accesses are shown below this point; the leaf of each access call path is highlighted with a rectangle. One access accounts for 19.3% of total remote memory accesses and the other for 2.9%. One can select a highlighted access to display its source code in the top pane. Insets in Figure 4 show the source code for these accesses. `S_diag_j` is accessed in different loops. Since these loops are inside OpenMP outlined functions (with suffix `$$OL$$`), they are executed by multiple threads in a parallel region.

The performance data in the GUI shows that there is a mismatch between the allocation of `S_diag_j` and initialization by the master thread in one NUMA domain, and accesses by OpenMP worker threads executing in other NUMA domains. The workers all compete for memory bandwidth to access the data in the master's NUMA domain. Besides `S_diag_j`, there are many other variables in AMG2006 with the same NUMA problem. To avoid contending for data allocated in a single NUMA domain, we launch the program with `numactl` [18, 19] and specify that all memory allocations should be interleaved across all NUMA domains. Table 2 shows the performance improvement. Using `numactl` reduces the running time of the solver phase from 105s to 87s. However, with `numactl` the initialization and setup phases are slower because interleaved allocations are more costly. The higher cost of interleaved allocation offsets its benefits for the solver phase.

phases	initialization	setup	solver	whole program
original	26s	420s	105s	551s
numactl	52s	426s	87s	565s
libnuma	28s	421s	80s	529s

Table 2: Improvement for different phases of AMG2006 using coarse-grained numactl and fine-grained libnuma.

A more surgical approach is to apply `libnuma`'s interleaved allocator [18] only to problematic variables identified by HPCToolkit. We used the bottom-up view provided by HPCToolkit's GUI, shown in Figure 5, to identify problematic variables. This view shows different call sites that invoke the hypre allocator. Figure 5 shows that while `S_diag_j` accounts for 22.2% of remote accesses, there are other 6 vari-

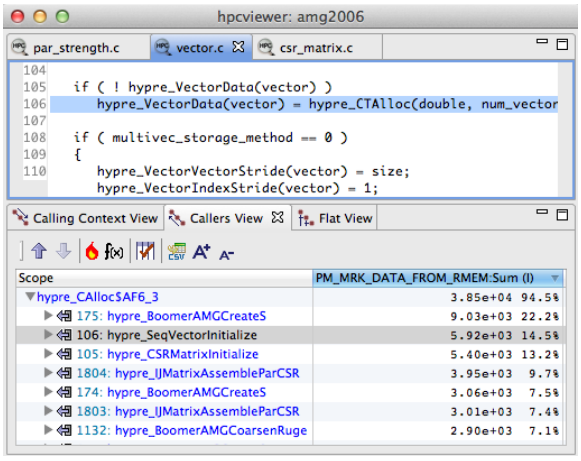

Figure 5: The bottom-up data-centric view of AMG2006 shows problematic allocation call sites which may reside in different call paths.

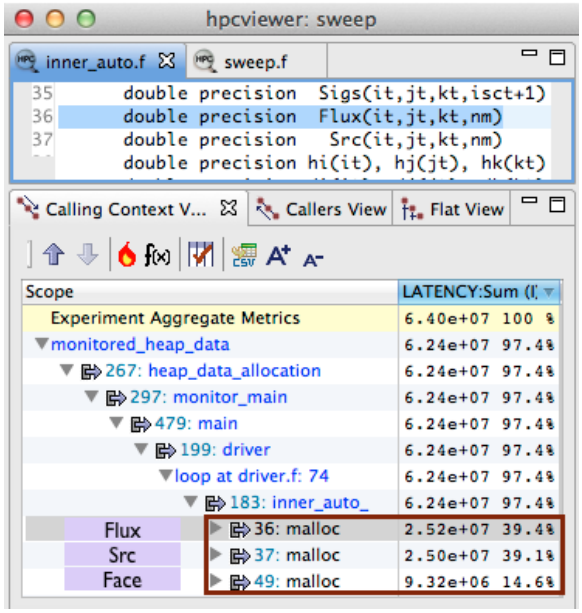

Figure 6: The data-centric view of Sweep3D shows heap allocated variables with high latency. Three highlighted arrays have high memory access latency.

ables that are the target of more than 7% of remote accesses. We focus on these problematic variables. If a variable is allocated and initialized by the master thread, we use `libnuma`'s interleaved allocator instead. If a variable is initialized in parallel, we change the `calloc` to `malloc`, which enables the "first-touch" policy to allocate memory near the computation. Table 2 shows that using `libnuma` to apply interleaved allocation selectively avoids dilating the cost of the setup and initialization phases. Moreover, the execution time of the solver phase is 8% faster than when using `numactl` and ubiquitous interleaved allocation. Using `libnuma`, we avoid interleaved allocation for thread local data, which eliminates remote accesses to that data that arise with ubiquitous interleaved allocation.

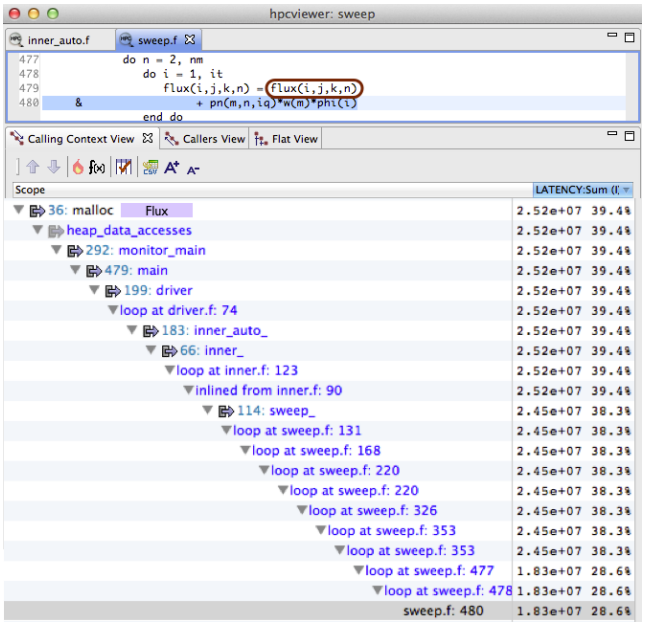
hpcviewer: sweep
inner_auto.f
sweep.f
do n = 2, nm
do i = 1, it
flux(i,j,k,n) =
+ pn(m,n,iq)*w(m)*phi(i)
end do
Calling Context View
Callers View
Flat View
Scope
LATENCY:Sum (I
36: malloc Flux 2.52e+07 39.4%
heap_data_accesses 2.52e+07 39.4%
292: monitor_main 2.52e+07 39.4%
479: main 2.52e+07 39.4%
199: driver 2.52e+07 39.4%
loop at driver.f: 74 2.52e+07 39.4%
183: inner_auto_ 2.52e+07 39.4%
66: inner_ 2.52e+07 39.4%
loop at inner.f: 123 2.52e+07 39.4%
inlined from inner.f: 90 2.52e+07 39.4%
114: sweep_ 2.45e+07 38.3%
loop at sweep.f: 131 2.45e+07 38.3%
loop at sweep.f: 168 2.45e+07 38.3%
loop at sweep.f: 220 2.45e+07 38.3%
loop at sweep.f: 220 2.45e+07 38.3%
loop at sweep.f: 326 2.45e+07 38.3%
loop at sweep.f: 353 2.45e+07 38.3%
loop at sweep.f: 353 2.45e+07 38.3%
loop at sweep.f: 477 1.83e+07 28.6%
loop at sweep.f: 478 1.83e+07 28.6%
sweep.f: 480 1.83e+07 28.6%

Figure 7: The data-centric view of Sweep3D shows a memory access with long latency to array Flux. This access is in a deep call chain.

5.2 Sweep3D

We ran Sweep3D on our 48-core AMD system and used IBS to monitor data fetch latency, which highlights data locality issues. The second line of the navigation pane in Figure 6 shows that 97.4% of total latency is associated with heap allocated variables. The top three heap allocated variables, `Flux`, `Src`, and `Face` account for 39.4%, 39.1%, and 14.6% of the latency respectively. Because these three arrays account for 93.1% of total latency, we only focus on optimizing their locality. Figure 7 shows a problematic access to `Flux`, which accounts for 28.6% of the total latency. This access, residing in line 480, is deeply nested in the call chain and loops. The two inner-most loops in line 477 and 478 traverse `Flux` with left-most dimension first and right-most dimension second. Since Fortran uses column-major array layouts, the loop in line 477 has a long access stride. These long strides disrupt spatial locality and hardware prefetching, which leads to elevated cache and TLB miss rates. To improve data locality, one could consider changing the access pattern by interchanging loops or transforming the data layout. In this case, interchanging loops is problematic. Examining the rest of the accesses to `Flux`, we see all are problematic, so we interchange the dimensions of `Flux` by inserting the last dimension between the first and second. With this data transformation, accesses to `Flux` have unit stride and improved spatial locality.

Both `Src` and `Face` suffer from the same spatial locality problem as `Flux`. Similarly, we transpose their layouts to match their access patterns. The optimization reduces the execution time of the whole program by 15%. It is worth noting that marked event sampling on POWER7 can also identify such optimization opportunities. One can sample PM_MRK_DATA_FROM_L3 event to quantify the locality issue in Sweep3D. Because Sweep3D is a pure MPI program, no NUMA problem exists because MPI processes are always

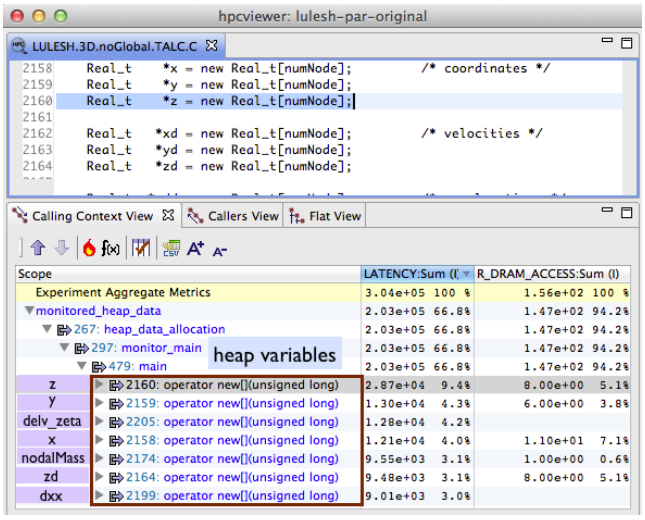
hpcviewer: lulesh-par-original
LULESH.3D.noGlobal.TALC.C
Real_t *x = new Real_t[numNode]; /* coordinates */
Real_t *y = new Real_t[numNode];
Real_t *z = new Real_t[numNode];
Real_t *xd = new Real_t[numNode]; /* velocities */
Real_t *yd = new Real_t[numNode];
Real_t *zd = new Real_t[numNode];
Calling Context View
Callers View
Flat View
Scope
LATENCY:Sum (I
R_DRAM_ACCESS:Sum (I)
Experiment Aggregate Metrics 3.04e+05 100 % 1.56e+02 100 %
monitored_heap_data 2.03e+05 66.8% 1.47e+02 94.2%
267: heap_data_allocation 2.03e+05 66.8% 1.47e+02 94.2%
297: monitor_main 2.03e+05 66.8% 1.47e+02 94.2%
479: main 2.03e+05 66.8% 1.47e+02 94.2%
heap variables
z 2160: operator new[](unsigned long) 2.87e+04 9.4% 8.00e+00 5.1%
y 2159: operator new[](unsigned long) 1.30e+04 4.3% 6.00e+00 3.8%
delv_zeta 2205: operator new[](unsigned long) 1.28e+04 4.2%
x 2158: operator new[](unsigned long) 1.21e+04 4.0% 1.10e+01 7.1%
nodalMass 2174: operator new[](unsigned long) 9.55e+03 3.1% 1.00e+00 0.6%
zd 2164: operator new[](unsigned long) 9.48e+03 3.1% 8.00e+00 5.1%
dxx 2199: operator new[](unsigned long) 9.01e+03 3.0%

Figure 8: The data-centric view shows problematic heap allocated arrays with high latency in LULESH. The red rectangle encloses the call paths of all heap variables that incur significant memory latency.

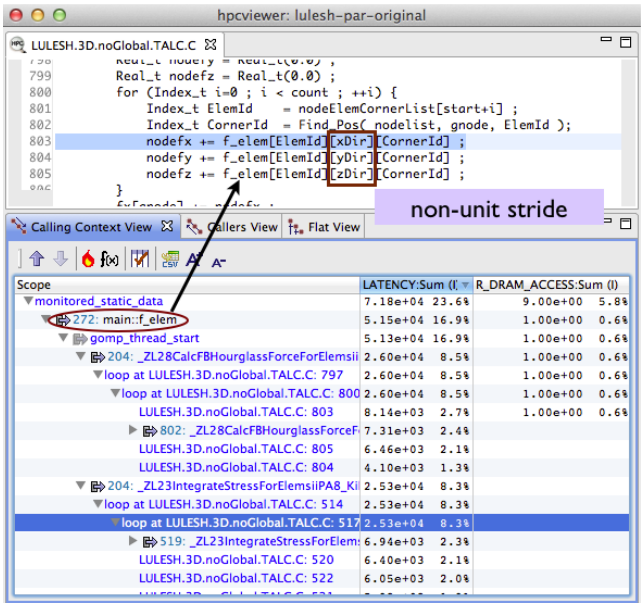
hpcviewer: lulesh-par-original
LULESH.3D.noGlobal.TALC.C
Real_t nodefz = Real_t(0.0) ;
for (Index_t i=0 ; i < count ; ++i) {
Index_t ElemId = nodeElemCornerList[start+i] ;
Index_t CornerId = Find_Pos(nodelist, gnode, ElemId);
nodefx += f_elem[ElemId][xDir][CornerId] ;
nodefy += f_elem[ElemId][yDir][CornerId] ;
nodefz += f_elem[ElemId][zDir][CornerId] ;
}
non-unit stride
Calling Context View
Callers View
Flat View
Scope
LATENCY:Sum (I
R_DRAM_ACCESS:Sum (I)
monitored_static_data 7.18e+04 23.6% 9.00e+00 5.8%
272: main::f_elem 5.15e+04 16.9% 1.00e+00 0.6%
gomp_thread_start 5.13e+04 16.9% 1.00e+00 0.6%
204: _ZL28CalcFBHourglassForceForElemsii 2.60e+04 8.5% 1.00e+00 0.6%
loop at LULESH.3D.noGlobal.TALC.C: 797 2.60e+04 8.5% 1.00e+00 0.6%
loop at LULESH.3D.noGlobal.TALC.C: 800 2.60e+04 8.5% 1.00e+00 0.6%
LULESH.3D.noGlobal.TALC.C: 803 8.14e+03 2.7% 1.00e+00 0.6%
802: _ZL28CalcFBHourglassForceF 7.31e+03 2.4%
LULESH.3D.noGlobal.TALC.C: 805 6.46e+03 2.1%
LULESH.3D.noGlobal.TALC.C: 804 4.10e+03 1.3%
204: _ZL23IntegrateStressForElemsiiPA8_Ki 2.53e+04 8.3%
loop at LULESH.3D.noGlobal.TALC.C: 514 2.53e+04 8.3%
loop at LULESH.3D.noGlobal.TALC.C: 517 2.53e+04 8.3%
519: _ZL23IntegrateStressForElems 6.94e+03 2.3%
LULESH.3D.noGlobal.TALC.C: 520 6.40e+03 2.1%
LULESH.3D.noGlobal.TALC.C: 522 6.05e+03 2.0%

Figure 9: The data-centric view shows a problematic static variable and its accesses with high latency in LULESH. The ellipse shows that variable's name.

co-located with their data and thus there is no need to examine NUMA-related events.

5.3 LULESH

We ran LULESH on our 48-core AMD system. As with Sweep3D, we used IBS for data-centric monitoring. The first metric column in Figure 8 shows the data access latency associated with heap allocated variables in LULESH. The second metric column shows a NUMA-related metric that reflects accesses to remote DRAM; this event is analogous to the POWER7 marked event PM_MRK_DATA_FROM_RMEM. The figure shows that heap allocated variables account for 66.8% of total latency and 94.2% of the execution's remote memory accesses. The individual heap allocated variables are shown along the allocation call path. Annotations to the left of the allocation call sites show the names of the variables allocated. Each of the top seven heap allocated variables accounts for 3.0–9.4% of the total latency. The R_DRAM_ACCESS metric shows that most of these variables are accessed remotely. By examining the source code, we found that all heap allocated variables in LULESH are allocated and initialized by the master thread. According to the Linux "first touch" policy, all of these variables are allocated in the memory attached to the NUMA node containing the master thread. Consequently, the memory bandwidth of that NUMA node becomes a performance bottleneck. To alleviate contention for that bandwidth, we use `libnuma` to allocate all variables with high remote accesses in an interleaved fashion. This optimization speeds up the program by 13%.

We continue our analysis of LULESH by considering static variables. Figure 9 shows that static variables account for 23.6% of total access latency. The static variable `f_elem` is a hotspot because it accounts for 17% of total latency. There are two principal loops that have high access latency for `f_elem`. Both loops have exactly the same structure and one is shown in the source pane of Figure 9. From the figure, one can see that `f_elem` is a three-dimensional array. The first dimension (left-most) is an indirect access using array `nodeElemCornerList` in line 801. The last dimension (right-most) is computed from a function `Find_Pos` in line 802. Thus, accesses to `f_elem` are irregular. Though optimizing data locality for irregular accesses is difficult, we found one opportunity to enhance the data locality for `f_elem` in this loop. The second dimension (highlighted by a rectangle) ranges from 0 to 2. We transposed `f_elem` to make this dimension the last, which enables these three accesses to exploit spatial locality since C is row-major. This transposition reduces LULESH's execution time by 2.2%.

5.4 Streamcluster

We ran Streamcluster on a node of our POWER7 system with 128 threads. Streamcluster suffers from serious NUMA data locality issues. Figure 10 shows that 98.2% of total remote memory accesses are related to heap allocated variables. The annotation to the left of the allocation that accounts for 92.6% of total remote accesses shows that it is associated with the variable `block`. Line 175 contains problematic accesses to `p1.coord` and `p2.coord`, which use pointers to access regions of `block`. This code is called from two different OpenMP parallel regions and accounts for 55.5% and 37% of total remote accesses from these two contexts. Examining the source code, we found that `block` is allocated and initialized by the master thread, so all worker threads access it remotely. We address this problem by leveraging the Linux "first touch" policy. Initializing `block` in parallel allocates parts of it near each thread using it. The optimization reduces both remote accesses and contention for memory bandwidth. We also applied this optimization to `point.p`, which accounts for 5.5% remote accesses. This optimization reduces Streamcluster execution time by 28%.

5.5 Needleman-Wunsch

We ran Needleman-Wunsch on a node of our POWER7

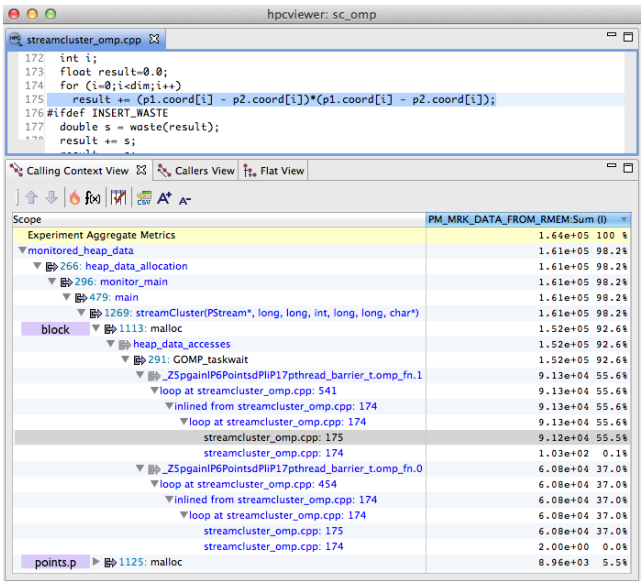
hpcviewer: sc_omp
streamcluster_omp.cpp
172 int i;
173 float result=0.0;
174 for (i=0;i<dim;i++)
175 result += (p1.coord[i] - p2.coord[i])*(p1.coord[i] - p2.coord[i]);
176 #ifdef INSERT_WASTE
177 double s = waste(result);
result += s;
Calling Context View
Callers View
Flat View
Scope
PM_MRK_DATA_FROM_RMEM:Sum (I)
Experiment Aggregate Metrics 1.64e+05 100 %
monitored_heap_data 1.61e+05 98.2%
266: heap_data_allocation 1.61e+05 98.2%
296: monitor_main 1.61e+05 98.2%
479: main 1.61e+05 98.2%
1269: streamCluster(PStream*, long, long, int, long, long, char*) 1.61e+05 98.2%
block 1113: malloc 1.52e+05 92.6%
heap_data_accesses 1.52e+05 92.6%
291: GOMP_taskwait 1.52e+05 92.6%
_Z5pgainlP6PointsdPliP17pthread_barrier_t.omp_fn.1 9.13e+04 55.6%
loop at streamcluster_omp.cpp: 541 9.13e+04 55.6%
inlined from streamcluster_omp.cpp: 174 9.13e+04 55.6%
loop at streamcluster_omp.cpp: 174 9.13e+04 55.6%
streamcluster_omp.cpp: 175 9.12e+04 55.5%
streamcluster_omp.cpp: 174 1.03e+02 0.1%
_Z5pgainlP6PointsdPliP17pthread_barrier_t.omp_fn.0 6.08e+04 37.0%
loop at streamcluster_omp.cpp: 454 6.08e+04 37.0%
inlined from streamcluster_omp.cpp: 174 6.08e+04 37.0%
loop at streamcluster_omp.cpp: 174 6.08e+04 37.0%
streamcluster_omp.cpp: 175 6.08e+04 37.0%
streamcluster_omp.cpp: 174 2.00e+00 0.0%
points.p 1125: malloc 8.96e+03 5.5%

Figure 10: The data-centric view associates a large number of NUMA related events to a problematic heap allocated variable and its inefficient accesses in Streamcluster benchmark.

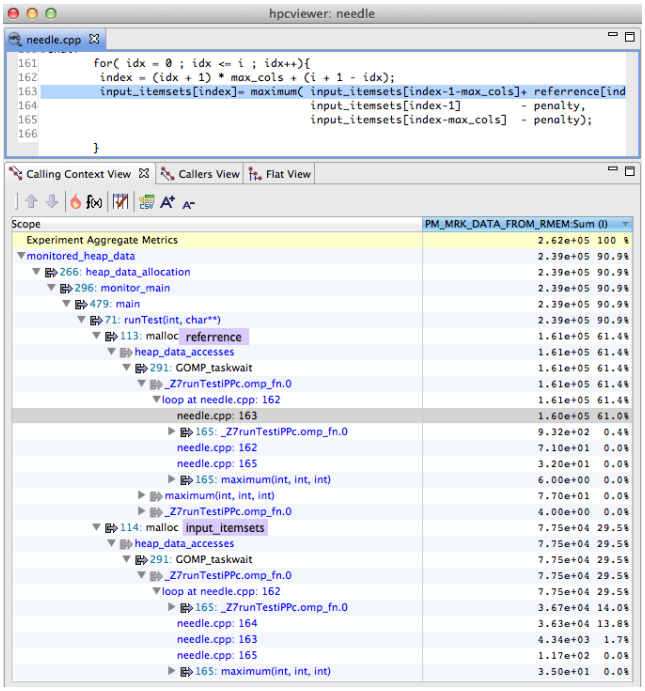
hpcviewer: needle
needle.cpp
161 for(idx = 0 ; idx <= i ; idx++){
162 index = (idx + 1) * max_cols + (i + 1 - idx);
163 input_itemsets[index]= maximum(input_itemsets[index-1-max_cols]+ referrence[ind
164 input_itemsets[index-1] - penalty,
165 input_itemsets[index-max_cols] - penalty);
166
}
Calling Context View
Callers View
Flat View
Scope
PM_MRK_DATA_FROM_RMEM:Sum (I)
Experiment Aggregate Metrics 2.62e+05 100 %
monitored_heap_data 2.39e+05 90.9%
266: heap_data_allocation 2.39e+05 90.9%
296: monitor_main 2.39e+05 90.9%
479: main 2.39e+05 90.9%
71: runTest(int, char**) 2.39e+05 90.9%
113: malloc referrence 1.61e+05 61.4%
heap_data_accesses 1.61e+05 61.4%
291: GOMP_taskwait 1.61e+05 61.4%
_Z7runTestiPPc.omp_fn.0 1.61e+05 61.4%
loop at needle.cpp: 162 1.61e+05 61.4%
needle.cpp: 163 1.60e+05 61.0%
165: _Z7runTestiPPc.omp_fn.0 9.32e+02 0.4%
needle.cpp: 162 7.10e+01 0.0%
needle.cpp: 165 3.20e+01 0.0%
165: maximum(int, int, int) 6.00e+00 0.0%
maximum(int, int, int) 7.70e+01 0.0%
_Z7runTestiPPc.omp_fn.0 4.00e+00 0.0%
114: malloc input_itemsets 7.75e+04 29.5%
heap_data_accesses 7.75e+04 29.5%
291: GOMP_taskwait 7.75e+04 29.5%
_Z7runTestiPPc.omp_fn.0 7.75e+04 29.5%
loop at needle.cpp: 162 7.75e+04 29.5%
165: _Z7runTestiPPc.omp_fn.0 3.67e+04 14.0%
needle.cpp: 164 3.63e+04 13.8%
needle.cpp: 163 4.34e+03 1.7%
needle.cpp: 165 1.17e+02 0.0%
165: maximum(int, int, int) 3.50e+01 0.0%

Figure 11: The data-centric view highlights two problematic heap allocated variables and their inefficient accesses in Needleman-Wunsch benchmark.

system with 128 threads. As with Streamcluster, this code suffers from a high ratio of remote memory accesses. A snapshot of HPCToolkit's GUI shown in Figure 11 indicates that 90.9% of remote memory accesses are associated with heap allocated variables. Two variables, `referrence` and `input_itemsets`, are hot spots that account for 61.4% and 29.5% of total remote accesses. The problematic accesses occur on lines 163–165. The `maximum` function called inside an OpenMP parallel region `_Z7runTestiPPc.omp_fn.0` takes both `referrence` and `input_itemsets` as inputs. However, both variables are allocated and initialized by the master thread. To address this problem, we use `libnuma` to distribute the allocation of these two variables across all NUMA nodes to alleviate contention for memory bandwidth. This optimization speeds up the program by 53%.

6. RELATED WORK

Unlike HPCTookit's integrated view of all data locality problems, prior work on measurement-based data-centric profilers only focuses on part of the problem. Section 6.1 describes tools that pinpoint poor temporal and spatial cache locality bottlenecks; Section 6.2 discusses tools that identify NUMA-related bottlenecks in threaded programs.

6.1 Tools for identifying poor cache locality

Irvin and Miller were perhaps the first to recognize the importance of data for performance [16]. In response, they extended Paradyn [30] to support a data view. They used Paradyn to dynamically instrument an executable to measure performance of semantic operations, such as collective communication primitives, and they used static analysis to associate measurements with data structures. Since they measure using instrumentation rather than hardware counters, details of hardware behavior (e.g. cache misses or latency) are unobservable.

In the first work to employ asynchronous, event-based sampling (EBS) for data-centric analysis, Itzkowitz et al. introduced memory profiling to Sun ONE Studio [17]. They apply this tool to collect and analyze memory accesses in a sequential program and report measurement data in flat profiles. Code-centric and data-centric views are not related.

Buck and Hollingsworth developed Cache Scope [7] to perform data-centric analysis using Itanium 2 event address registers (EAR). Cache Scope associates latency with data objects and functions that accessed them. Unlike HPCToolkit, Cache Scope does not collect information about calling context and only associates latency metrics with code at the procedure level rather than at the level of loops or individual source lines.

In previous work [25, 26], we extended HPCToolkit to use IBS to analyze memory hierarchy performance. This work focused exclusively on sequential programs and as a result did not consider NUMA-related performance problems. Furthermore, this work didn't support attribution of memory hierarchy performance for static variables.

Tallent and Kerbyson also extended HPCToolkit to support data-centric tracing [37]. However, their work focused on global arrays used in the Partitioned Global Address Space (PGAS) programming models, rather than heap-allocated or static variables used in a common program.

A common drawback of these tools is that, to our knowledge, none of them supports measurement and analysis of scalable parallel programs with low overhead. Though these tools can identify locality bottlenecks in sequential programs, parallel versions (either threaded, MPI, or hybrid) of these programs may suffer from different performance problems, particularly with multithreaded programs running on nodes with a NUMA memory hierarchy.

6.2 Tools for identifying NUMA inefficiencies

Memphis [28] uses IBS for low-overhead collection of NUMA-related performance information on AMD Opteron-based multicore platforms. Memphis collects information about accesses to remote cache or memory and then associates these events with static data variables. Unlike HPCToolkit, Memphis doesn't have mechanisms for attributing access costs back to dynamic allocations and doesn't attribute costs to full calling contexts.

MemProf [20] is another data-centric tool based on AMD IBS, for analyzing NUMA problems. It maps IBS samples to both static and heap allocated variables. Unlike HPCToolkit, MemProf records a trace of each IBS sample and variable allocation rather than collapsing it on-the-fly into a compact profile. The resulting high data volume makes this problematic to scale to a cluster with a large number of nodes. Moreover, MemProf does not map performance metrics to individual static variables; instead, it treats all static variables from a load module as one group and coarsely attributes metrics to these groups.

It is not clear that either of these tools can be applied to study MPI+OpenMP programs on a cluster. Additionally, these tools exclusively focus on NUMA locality problems. Furthermore, neither of these tools supports precise attribution of metrics to both static and heap allocated variables.

7. CONCLUSION AND FUTURE WORK

Augmenting HPCToolkit with support for data-centric profiling of highly parallelized programs enables it to quantify temporal, spatial, and NUMA locality. HPCToolkit's data-centric measurement and analysis capabilities leverage PMU hardware on IBM POWER7 and AMD Opteron processors. After the experimental work reported in this paper was completed, we extended HPCToolkit to support data-centric analysis using PMU hardware in additional processors, including Intel's Itanium2 and Ivy Bridge. By using sampling and a compact profile representation, HPCToolkit has low time and space overhead; this helps it scale to a large number of cores. In case studies, we applied HPCToolkit's data-centric analysis capabilities to study five well-known parallel benchmarks and attribute data-centric metrics to both code and variables. These capabilities provided intuitive analysis results that enabled us to easily identify and optimize data locality bottlenecks. With data-centric feedback from HPCToolkit, we were able to improve the performance of these benchmarks by 13–53%.

Hardware sampling support for data-centric profiling is insufficient in today's supercomputers. For example, ORNL's Titan supercomputer uses AMD processors that support instruction-based sampling. However, the operating system on Titan disables IBS. LLNL's Sequoia supercomputer is based on Blue Gene/Q processors. Though the A2 cores of a Blue Gene/Q ASIC have SIAR and SDAR registers as part of support for instruction sampling, this capability is not usable because full support for instruction sampling is missing from the multicore ASIC. The utility of measurement-based data-centric analysis of parallel program executions, which we demonstrate, motivates including hardware support for data-centric measurement in future generation systems.

We plan to extend our work in several ways. First, rather than overlooking heap allocations smaller than 4K, we think that monitoring some of them will enable HPCToolkit to provide useful data-centric feedback for programs with data structures built from lots of small allocations. Second, we plan to explore extensions that enable us to associate data-centric measurements with stack-allocated variables. Finally, we plan to enhance HPCToolkit's measurement and analysis to provide guidance for where and how to improve data locality by pinpointing initializations that associate data with a memory module and identifying opportunities to apply transformations such as data distribution, array regrouping, and loop fusion.

Acknowledgements

This work was supported in part by subcontract B602160 from Lawrence Livermore National Laboratory.

8. REFERENCES

[1] Accelerated Strategic Computing Initiative. The ASCI Sweep3D Benchmark Code. `http://wwwc3.lanl.gov/pal/software/sweep3d`, 2009.

[2] L. Adhianto et al. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22:685–701, 2010.

[3] Advanced Micro Devices. AMD CodeAnalyst performance analyzer. `http://amddevcentral.com/tools/hc/CodeAnalyst/Pages/default.aspx`.

[4] J. M. Anderson et al. Continuous profiling: where have all the cycles gone? *ACM TOCS.*, 15(4):357–390, 1997.

[5] K. Beyls and E. D'Hollander. Discovery of locality-improving refactorings by reuse path analysis. *Proc. of the* 2^{nd} *Intl. Conf. on High Performance Computing and Communications (HPCC)*, 4208:220–229, Sept. 2006.

[6] K. Beyls and E. H. D'Hollander. Refactoring for data locality. *Computer*, 42(2):62 –71, Feb. 2009.

[7] B. R. Buck and J. K. Hollingsworth. Data centric cache measurement on the Intel ltanium 2 processor. In *SC '04: Proc. of the 2004 ACM/IEEE Conf. on Supercomputing*, page 58, Washington, DC, USA, 2004. IEEE Computer Society.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. of the 2009 IEEE Intl. Symp. on Workload Characterization (IISWC)*, pages 44–54, Washington, DC, USA, 2009.

[9] J. Dean et al. ProfileMe: Hardware support for instruction-level profiling on out-of-order processors. In *Proc. of the* 30^{th} *annual ACM/IEEE Intl. Symposium on Microarchitecture*, pages 292–302, Washington, DC, USA, 1997.

[10] P. J. Drongowski. Instruction-based sampling: A new performance analysis technique for AMD family 10h processors. `http://developer.amd.com/Assets/AMD_IBS_paper_EN.pdf`, November 2007.

[11] N. Froyd, J. Mellor-Crummey, and R. Fowler. Low-overhead call path profiling of unmodified, optimized code. In *Proc. of ICS'05*, pages 81–90, New York, NY, USA, 2005. ACM Press.

[12] S. L. Graham, P. B. Kessler, and M. K. McKusick. Gprof: A call graph execution profiler. In *Proc. of the*

1982 SIGPLAN Symposium on Compiler Construction, pages 120–126, New York, NY, USA, 1982. ACM Press.

[13] Intel VTune Amplifier XE 2013. `http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe`, April 2013.

[14] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual, Volume 3B: System programming guide, Part 2, Number 253669-032US. `http://www.intel.com/Assets/PDF/manual/253669.pdf`, June 2010.

[15] Intel Corporation. Intel Itanium Processor 9300 series reference manual for software development and optimization. `http://www.intel.com/Assets/PDF/manual/323602.pdf`, March 2010.

[16] R. B. Irvin and B. P.Miller. Mapping performance data for high-level and data views of parallel program performance. In *Proc. of ICS'96*, pages 69–77, New York, NY, USA, 1996. ACM.

[17] M. Itzkowitz, B. J. N. Wylie, C. Aoki, and N. Kosche. Memory profiling using hardware counters. In *Proc. of the 2003 ACM/IEEE Conf. on Supercomputing*, page 17, Washington, DC, USA, 2003. IEEE Computer Society.

[18] A. Kleen. A NUMA API for Linux. `http://developer.amd.com/wordpress/media/2012/10/LibNUMA-WP-fv1.pdf`, 2005.

[19] A. Kleen. nuamctl – Linux man page. `http://linux.die.net/man/8/numactl`, 2005.

[20] R. Lachaize, B. Lepers, and V. Quéma. Memprof: a memory profiler for NUMA multicore systems. In *Proceedings of the 2012 USENIX Annual Technical Conf.*, USENIX ATC'12, Berkeley, CA, USA, 2012.

[21] Lawrence Livermore National Laboratory. Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH). `https://codesign.llnl.gov/lulesh.php`.

[22] Lawrence Livermore National Laboratory. LLNL Sequoia Benchmarks. `https://asc.llnl.gov/sequoia/benchmarks`.

[23] A. R. Lebeck and D. A. Wood. Cache profiling and the SPEC benchmarks: A case study. *Computer*, 27(10):15–26, 1994.

[24] J. Levon *et al.* OProfile. `http://oprofile.sourceforge.net`.

[25] X. Liu and J. Mellor-Crummey. Pinpointing data locality problems using data-centric analysis. In *Proc. of CGO'11*, pages 171–180, Washington, DC, 2011.

[26] X. Liu and J. Mellor-Crummey. Pinpointing data locality bottlenecks with low overheads. In *Proc. of ISPASS 2013*, Austin, TX, USA, April 21-23, 2013.

[27] M. Martonosi, A. Gupta, and T. Anderson. Memspy: analyzing memory system bottlenecks in programs. *SIGMETRICS Perform. Eval. Rev.*, 20(1):1–12, 1992.

[28] C. McCurdy and J. S. Vetter. Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In *IEEE Intl. Symp. on Performance Analysis of Systems Software (ISPASS)*, pages 87–96, Mar. 2010.

[29] Message Passing Interface Forum. MPI: A message passing interface standard. `http://www.mcs.anl.gov/research/projects/mpi`, 2013.

[30] B. P. Miller et al. The Paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.

[31] OpenMP Architecture Review Board. OpenMP application program interface, version 3.0. `http://www.openmp.org/mp-documents/spec30.pdf`, May 2008.

[32] A. Rane and J. Browne. Enhancing performance optimization of multicore chips and multichip nodes with data structure metrics. In *Proc. of the 12th Intl. Conf. on Parallel Architectures and Compilation Techniques*, Minneapolis, MN, USA, 2012. IEEE Computer Society.

[33] Rogue Wave Software. ThreadSpotter manual, version 2012.1. `http://www.roguewave.com/documents.aspx?EntryId=1492`, August 2012.

[34] D. L. Schuff, M. Kulkarni, and V. S. Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proc. of PACT'10*, PACT '10, pages 53–64, New York, NY, USA, 2010. ACM.

[35] M. Srinivas et al. IBM POWER7 performance modeling, verification, and evaluation. *IBM JRD*, 55(3):4:1–4:19, May-June 2011.

[36] N. R. Tallent, L. Adhianto, and J. M. Mellor-Crummey. Scalable identification of load imbalance in parallel executions using call path profiles. In *Proc. of Intl. Conf. for High-Performance Computing, Networking, Storage and Analysis*, New York, NY, USA, November 2010. ACM.

[37] N. R. Tallent and D. Kerbyson. Data-centric performance analysis of PGAS applications. In *Proc. of the Second Intl. Workshop on High-performance Infrastructure for Scalable Tools (WHIST)*, San Servolo Island, Venice, Italy, 2012.

[38] N. R. Tallent, J. Mellor-Crummey, and M. W. Fagan. Binary analysis for measurement and attribution of program performance. In *Proc. of the 2009 ACM PLDI*, pages 441–452, NY, NY, USA, 2009. ACM.

[39] Y. Zhong and W. Chang. Sampling-based program locality approximation. In *Proc. of the 7th Intl. Symposium on Memory Management*, ISMM '08, pages 91–100, New York, NY, USA, 2008. ACM.